Writing Secure Software

It's often claimed that the biggest problem with security is that practitioners are unclear as to what the problem is. In summary, it's insecure computer software.

The best network firewall provides only minimal defense if it permits access to unreliable software. Moreover, any firewall (either appliance or program) is written in software.

Similarly, the strongest encryption algorithms may only permit attackers to securely communicate with insecure software.

> *"Using encryption on the Internet is the equivalent of arranging an armoured card to deliver credit-card information from someone living in a cardboard box, to someone living on a park bench."*
>
> Gene Spafford

Internet-enabled applications, including ones developed within companies, form the greatest category of security risks.

Each year, at least 80% of security advisories from CERT/CC (the Computer Emergency Response Team Coordination Center, *www.cert.org*) report security vulnerabilities caused by insecure software.

Moreover, again 80% of all vulnerabilities cannot be addressed using stronger encryption.

> *"Adding ever stronger encryption is akin to placing a Yale lock on a paper bag"*
>
> *Bruce Schneier*

Trends In Insecure Software

Not only are the number of (reported) security vulnerabilities themselves increasing, but so are the number due to insecure software. There are some clear trends:

- The growth and ubiquity of computer networking has increased both the number of avenues for attack, and the ease with which attacks may be made. Moreover, we observe an increasing use of automated attacks. Clearly, physical access is no longer a barrier, and distance is an advantage!

- The dramatic increases in software size and complexity, and the hyperexponential growth in their interactions. A desktop machine running a modern operating system (such as Windows-XP with 35 millions LOC) and various, 3rd-party, applications, cannot help but have an increasing number of bugs.

- The use of low-level programming languages, such as C/C++, that do not protect against common low-level attacks is actually on the increase, not decrease. Moreover, the use of autonomous embedded devices (with Internet-connectivity) will fuel the use of C/C++ for some time.

- The trend toward extensible software – applications permit 3rd-party plugins, operating systems encourage "on-demand" updates over networks, applications support a greater degree of mobile and scripting code,

- Ill-considered approaches to software security often makes software more vulnerable, and not less. Examples include the use of "home-grown", secret, encryption algorithms, and "rush-to-market" solutions to support new technologies, such as digital certifi cates and sandboxed languages – the "Internet-time" phenomena.

Software Security Through Software Patches

Many software vendors still consider software security to be an add-on feature.

Software products are rushed to market, and patches or updates (often at additional cost to customers!) are released after vulnerabilities are brought to the attention of software vendors.

A wealth of software-engineering reports show that, in economic terms, finding and removing bugs in software *before* its release is orders of magnitude cheaper and more effective than trying to fix systems after release.

However:

- Developers can only patch security problems which they know about. Attackers may find problems that are never reported to developers.
- Patches, too, are rushed to market as a result of commercial pressures, and new vulnerabilities are often introduced, or exposed, as an consequence.
- Patches attempt to quickly address symptoms, and cannot (quickly enough) address the underlying causes.
- Patches are often not applied, or not quickly enough. System administrators may be reluctant to patch “working” systems,
- Home-users will typically never see (nor understand) the reasons nor solutions to vulnerabilities, and patches are often not cumulative, and hence very large.

Security vulnerabilities, if considered properly, are *bugs*. Why are they not given due consideration? Perhaps shrink-wrapped software is the cause? Why do developers have indemnity?

Open Source .versus. Closed Source

“Today’s security woes are not dominated by the existence of bugs that might be discovered by open-source developers studying system source-code.”

Fred Schneider

Despite popular trends toward transparency, including the OpenSource movement (often sidetracked by the political and philosophical issues), most software vendors still embrace secrecy of their software.

This is unsurprising, as companies have vast amounts of intellectual property (the other IP) invested in their software.

However, secrecy is frequently used as an excuse for poor software, and should not be effective.

While releasing source code certainly assists attackers, *not* releasing source code introduces *security-by-obscurity*, and develops a false sense of security that vulnerabilities will not be detected.

It is a false belief that (poor) code, compiled and only released as binaries, remains secret (consider reverse-engineering, variously legal in many countries, including Australia).

- An embarrassing vulnerability was found in MS-Frontpage in April 2000. The constant encryption key *“Netscape engineers are weenies!”* was embedded in the released binary, to be quickly found by inspection of the binary.

Open Source .versus. Closed Source, *continued*

The alternative approach, full release of source-code for inspection, is often termed “the many-eyeballs phenomena”, the belief that vulnerabilities will be revealed because many will have pored over, and fully understood, your code.

> *“Given enough eyeballs, all bugs are shallow”*
> Eric Raymond, Cathederal and the Bazaar

However, people often falsely trust that open-source projects have received suffi cient security, and are thus more secure.

Far more software is written than read!.

Simply releasing software source code is *not* a panacea for secure software. In nearly all cases, only the original software developer (often solo) is the one with full knowledge of the purpose or intended workings of software.

Other software *users* do not diligently attempt to understand software. Moreover, the economics and excitement of *free* software drives its *use*, not its *inspection*.

- A great counter-example comes from Stanford’s Donald Knuth, who in 1981 offered $2.56 (“256 pennies is one hexadecimal dollar”) to the fi rst person to fi nd a bug in his released TeX source code (or any of his 12 textbooks), and doubles the reward for each new bug.

Are Open Source projects more secure?

Certainly not, but the following observations suggest why there are far fewer viruses in the Open Source environment:

- The Open Source (and Macintosh environments) have considerably less market share, making them less “interesting” (less of a target) for virus writers.

- Modern Open Source applications (even under the Gnome and KDE managers) have fewer inter-application dependencies, and hence perform fewer actions “behind the backs of the user”. In particular, applications store preferences in per-application locations, rather than a shared location (Registry).

- The available choice of applications are considerably more fragmented under LINUX than under Windows (e.g. compare the high uptake of Outlook under Windows with the variety of LINUX mail agents).

- The *average* Open Source user has a higher working knowledge of their platform, can anticipate a wider range of problems, and diagnose a wider range of symptoms.

- And most importantly – very few mail applications support the immediate execution of attachments/executables, let alone without fi rst asking.

Buffer Overflows

By far the most effective solution to understanding software vulnerabilities is to be aware of the most common sources of error, and knowing their effects.

By far, the greatest single software security threat is the *buffer overflow*, according to CERT/CC accounting for 50% of all major vulnerabilities in 1999 and growing).

The root cause behind the buffer overflow is that the C/C++ languages provide inadequate run-time memory protection. In particular, there are no bounds-checks on array or pointer references (although recent enhancements to some compilers are improving the situation).

```
#include <stdio.h>

int main(int argc, char *argv[])
{
    char buf[BUFSIZ];

    gets(buf);
    return(0);
}
```

```
In function ‘main’:
text+0x17): the ‘gets’ function is dangerous and should not be used.
```

There are also a number of unsafe string operations in the standard C library, including:

strcpy, strcat, sprintf, gets and *scanf*.

Use of these functions is strongly discouraged, and warnings about their use are increasingly reported by compilers and linkers:

Buffer Overflows, *continued*

Buffer overflows, reading or writing past the end of a fi xed memory-sized block, can cause a number of diverse, and often unanticipated, program behaviours:

- programs can act in strange ways (but not crash),
- programs can fail completely, but not at the exact point of the overflow, or (worst of all)
- programs can continue without any (noticeable) effects.

Security vulnerabilities are introduced, depending upon:

- how much data are written past the buffer’s bounds,
- what data, if any, are overwritten when a buffer is overfi lled,
- what data end up replacing the data that gets overwritten,
- whether the program continues and attempts to *read* data that are overwritten, and
- whether the program continues and attempts to *execute* data that are overwritten.

Buffer Overflows – a simple example

Consider the following simplifi ed, but representative example:

```
#include <stdio.h>

int main(int argc, char *argv[])
{
    char answer[24];
    int  am_i_chris  = (getuid() == 333);

    printf("Delete files? ");
    gets(answer);
    if(am_i_chris)
        ......
    return(0);
}
```

In this example, presenting any input, via *stdin*, longer than the size of *answer* (24 bytes), will result in the memory treated as the Boolean *am_i_chris* being overwritten.

If the 25th character presented to *answer* in non-null (i.e. non-zero), then the rest of the program may be tricked into undertaking a different execution. The vulnerability is easily exposed:

- particularly, if the naive check of the user-id remains in production code,
- if an attacker has access to the source code,
- if an attacker chooses to reverse-engineer the code, and
- if an attacker can determine where in memory certain variables reside, they can "set" them to almost any desired value.

As we now focus on a limited number of processor architectures and operating systems, buffer overfbw attacks are often crafted for these common platforms.

Stack Overflows

"On many C implementations it is possible to corrupt the execution stack by writing past the end of an array declared auto in a routine. Code that does this is said to smash the stack, and can cause return from the routine to jump to a random address. This can produce some of the most insidious data-dependent bugs known to mankind."
aleph1@underground.org

Far more interesting, and far less of a challenge to attackers, are *stack-overflows*, because there is always something critical to security on a process's stack – the process's *return address*. An attacker's agenda for a stack overfbw is:

- fi nd a stack-allocated (automatic) buffer that we can overfbw that allows us to overwrite the return address in a run-time stack frame,
- place hostile *executable code* in memory to which we can jump when the function we are attacking returns,
- write over the return address on the stack with a value that causes the program to jump to the hostile code.

The objective is to inject executable instructions into the memory of the process, and force their execution. Under LINUX the objective is to emit the instructions for `execl("/bin/sh","sh",0)`, which on an Intel processor looks like:

```
  u_char execshell[] =
   "\xeb\x24\x5e\x8d\x1e\x89\x5e\x0b\x33\xd2\x89\x56\x07\x89\x56\x0f"
   "\xb8\x1b\x56\x34\x12\x35\x10\x56\x34\x12\x8d\x4e\x0b\x8b\xd1\xcd"
   "\x80\x33\xc0\x40\xcd\x80\xe8\xd7\xff\xff\xff/bin/sh";
```

The details of stack-smashing are complicated, but are now well automated – see *.../units/231.317/reading/smashstack.txt*

Stack Overflows, *continued*

There remain a few challenges for the attacker, particularly if attacking the standard C string functions, such as *gets*, because these functions return whenever a *NULL* byte is seen, and the zero-byte or integer zero is very frequently seen in instruction sequences. No problem for the dedicated attacker; generate code for:

```
void exploit()
{
    char *s = "/bin/sh";
    execl(s, s, 0xff ^ 0xff);
}
```

Once an attacker can force the execution of another program, typically a shell or command interpreter, they can then issue almost any other command.

In particular, if stack-overflow attacks are made against *setuid-root* programs, the attacker easily gains root privilege on a machine.

Stack overflow attacks may be performed in a number of ways:

- by an authorized user, who is (legally) logged in already,
- by an attacker the network, who has already compromised a standard user's account, or
- by an attacker, over a network, sending the special overflowing input (instructions) over a socket connection to a vulnerable program.

The infamous attacks of the **1980's** against the standard UNIX email agent, *sendmail*, first exploited stack-overflow vulnerabilities.

It's about time that these common vulnerabilities were fixed!

Race Conditions

Perhaps second only to buffer overflows in providing software security vulnerabilities, are *race conditions*.

By naive definition, a race condition occurs when assumptions are made about how software executes and *interacts in a time-dependent manner*.

They are only possible in software environments where multiple threads of control or operating system processes (or even asynchronous event processing) interact. When the form of interaction in unanticipated, a security vulnerability may be exposed.

Race conditions are one of the few occasions where a seemingly deterministic program can behave in a seriously non-deterministic way.

Like buffer overflows, they are an increasing threat, this time because we are writing more multi-threaded, interacting, software.

The condition occurs when an assumption is believed to hold true for a longer period of time than it actually does – the time during which violating the assumption leads to incorrect behaviour, is termed the *window of vulnerability*.

Race Conditions – a trivial example

Consider the following code from a Java servlet:

```
public class Counter extends HttpServlet {
    int count = 0;
    public void doGet(HttpServletRequest in,
                      HttpServeletResponse out)
           throws ServetException, IOException {
          .....
        ++count;
        out.println(count + "hits");
    }
}
```

This simple code has a race condition, because the multithreaded servlet (programmer) assumes that the value of `count` will be the same when printed, as it was just after it was incremented. Another servlet thread could increment `count`, just before "our" printing.

Even the attempt:

```
out.print(++count + "hits");
```

won't correct the vulnerability, because the method call and the evaluation of its arguments are not *atomic operations*.

Instead, we must constrain all accesses to `count`, while keeping such periods as short as possible:

```
synchronized(this) {
    my_count = ++count;
}
out.print(my_count + "hits");
```

Time-of-check .versus. Time-of-use

Not every race condition occurs in multithreaded programs. More frequently, vulnerabilities are exposed in single-threaded programs because the programmer *expects* there to be no interaction with other programs.

However, if an attacker can introduce an *unexpected interaction* (from another program), the resulting race condition may give the attacker the privileges of the original program.

Multiple processes typically interact via *shared data*, and the best example of such sharing is via the file-system. File-based race conditions are the most commonly observed security-threatening problems, and they follow a typical pattern:

- a check is performed on the property of a resource, such as the existence of a file, access permissions, or its size,
- the result of the check needs be true at the time some following code (which assumes the result still valid) is executed,
- however, between the check and the executed code, the "conditions" of the check changes.

```
if(access(filename, W_OK) == 0) {
    f = fopen(filename, "w+");
    write_file_file(f);
}
else
    fprintf(stderr, "could not write\n");
```

Between the *time-of-check* and the *time-of-use*, the file referenced by `filename` may have changed!

Time-of-check .versus. Time-of-use, *continued*

`access()` determines if the indicated file may be accessed in the indicated way, by the *real* UID of the process.

A “text editor” that needs to run as root may need to perform this check, to bypass the permissions it has when running *set-uid* root, i.e. we do not wish to check based on its *effective* UID.

The *window of vulnerability* here is between the calls to `access()` and `fopen()`, which assume they are considering the same file.

Using the above code, an attacker may be able to have the editor edit a bogus file, resulting in an important file being overwritten – */etc/passwd* being a common target.

The easiest attack, here, is to use a *symbolic link* file, which creates a file that looks like any other file, but simply points to a “real” file. The attacker creates a dummy file (with his permissions), and then creates a symbolic link to it:

```
$ touch dummy
$ ln -s dummy pointer
```

Now, the attacker tells the text-editor program to edit the file `pointer`. The attacker’s goal is to perform a command sequence, such as the following, all within the window of vulnerability:

```
$ rm pointer; ln -s /etc/passwd pointer
```

If successful, the attack will overwrite the system password file.

Impossible? The attacker writes a C program to spawn the editor, perform the file-manipulation commands, and checks to see if the real password file was overwritten. Repeat until done.

Avoiding the Window of Vulnerability

The previous problem has been introduced because the check of the file’s attributes has been performed on a file that “may change”.

We must avoid system-call and library function calls which use file *names* (aliases) as parameters. Even the standard library functions, which can only be composed of multiple system calls, often have windows of vulnerability between checking a file and accessing it.

Instead, a *file-descriptor* or *file-handle* to an already opened file should be used. Before performing a `stat` on an existing file, `open` the file and perform an `fstat` using the returned file-descriptor.

To develop very secure code, we need develop sequences of code to replace many of the standard techniques. For example, to open an arbitrary file, we can:

- before opening the file, call `lstat` on its filename (which checks the attributes of any possible link, not the file to which it may point), saving the `stat` structure,
- `open` the filename (which may have been a link),
- `fstat` the resulting file-descriptor, saving its `stat` structure,
- compare the fields of the two `stat` structures, the `st_mode`, the `st_inode`, and the `st_dev`, to ensure that the file (or its link) have not changed.

It is possible to build very paranoid code sequences to reduce the possibility of *symlink attacks*. Most programming language libraries should consider this for their future standards.

Securely Accessing Temporary Files

Creating temporary files in a shared space such as */tmp* is a common practice.

However, the use of temporary files are subject to the same symlink attacks as standard files, but with the added problems that attackers may be able to guess your program's temporary file names.

Consider the standard function `mktemp`:

```
strcpy(filename, "/tmp/XXXXXX");
mktemp(filename);
fd = fopen(filename, O_RDWR, 0600);
```

The problem is that while *mktemp* generates a "random" filename, guaranteed not to exist, it does not actually create the file – hence a race condition.

Moreover (still), the method of replacing the values of `XXXXXX` is far from random – the first five X's are replaced with the process's PID, and the final X with the first available 'a', 'b', 'c', ... These are easily guessable.

Instead, we should replace such uses with:

```
strcpy(filename, "/tmp/XXXXXX");
fd = mkstemp(filename);
```

The file is opened with the `O_EXCL` flag, guaranteeing that when `mkstemp` returns successfully we are the only user.

Better still, would be for our application to generate its own temporary directory name, and place its own temporary files therein. And *never* close and re-open a shared temporary file!

Randomness and Non-Determinism

A fairly common source of software vulnerability is the misuse of random number generators.

An attacker, with often little effort, can determine or predict the stream of random numbers that software is using (or will use), and use that knowledge to launch an attack.

Random numbers are used in a wide variety of applications, including session-keys for cryptographic streams, and the "random" names of temporary files and directories.

Without access to physical devices to provide randomness, software must generate streams of random number themselves – the use of *pseudo-random number generators*, PRNGs.

Most (common) software simply employs the standard random-number generation facilities from common languages, such as C and Java, and assumes that their `random` or `rand` functions provide enough randomness that the numbers cannot be guessed.

However, the determinism of software makes the generation difficult. The most common PRNGs employ *linear congruential generation*:

$$X_{n+1} = (a\,X_n + b) \bmod c$$

where the three constants, a, b, and c (usually prime numbers), fully define the sequence of random numbers.

Such generators typically produce 32-bit numbers, and for any a, b, and c the sequences are deterministic. Attackers, knowing exactly which functions (and hence a, b, and c) are employed can typically "break" the stream in a few seconds of brute-force comparison.

Randomness and Non-Determinism, *continued*

Often, the problem is not simply with the simplicity of the PRNG function itself (and there are now many more complex ones available) but with the genuine *randomness*, or amount of *entropy*, in the initial *seeds* to PRNGs.

Often, applications will choose very simple seeds, ones that may simply be determined from the process's runtime environment. Typical examples include the process's PID, its PPID, or the time of the process's creation.

The downfall of the first implementations of SSL in Netscape's *navigator* browser, was the naive combination of PID, PPID, and time used to generate a session key for each SSL session.

Software that requests very long (once-off) passphrases, or just asks you to type characters for ten seconds, do not really care *what* you type, but use the inter-key timings to generate some entropy.

```
fp = fopen("/dev/random", "r");
want = 64;
i    = 0;
while(want > 0) {
    fread(&ints[i++], sizeof(int), 1, fp);
    --want;
}
fclose(fp);
```

Both LINUX and Windows now provide access to system-calls or pseudo devices that generate streams of pseudo-random numbers, based on some hardware characteristics, such as the number of device interrupts over a recent interval.

Security aspects of web browsers and servers

It is often argued that the introduction of web browsers and servers has not really introduced any new types of security problems, just repeated the problems of olde, but now on a global scale.

Here we examine some security problems exposed by browsers and servers.

The Common Gateway Interface (CGI)

The resource named in a browser's HTTP request may refer to a program. HTTP does not specify how the server determines this, but it is considered good practice to limit their location to special directories containing only programs (and not data).

It is just historical convention that `/cgi-bin/` is a directory alias containing executable programs, typically written in scripting languages such as Perl, Python, Tcl, or maybe C/C++.

The only information available to a CGI program must come from the client's request (text) stream. Primarily, this means the resource name in the first request line, and the MIME header lines that follow.

There are two conventions for passing input to CGI programs:

- in the "CGI" convention, the query string is passed as the first command line parameter to the program.
- in HTML "Forms", the query string contains a sequence of varname=value pairs separated by "&", as in `"Name=Latham&Occupation=Aspirant"`.

Of note, all URLs requested from a website are typically logged. Clearly, passwords should not be transmitted this way!

Security problems with CGI scripts

Common Gateway Interface (CGI) scripts introduce problems as each script potentially exposes a new, or repeated, opportunity for exploitable bugs.

CGI scripts need be written with the same care and attention given to traditional Internet servers themselves, because, in fact, they are miniature servers. Unfortunately, for many Web authors, CGI scripts are their first encounter with network programming.

In general, CGI scripts present two major security problems:

- They may intentionally, or unintentionally, leak information about the host system, which may assist attackers.
- Scripts that process remote user input, such as the contents of an HTML web-form, may be vulnerable to attacks in which the remote user tricks them into executing commands.

Consider a basic script which attempts to email a requested file to a remote user. A too common technique is to obtain the requires filename and email address from web-form variables:

```
  mail $EMAILADDR < $FILENAME
```

where `EMAILADDR` and `FILENAME` are extracted from a CGI's input parameters, and the whole script is sent to a shell for simple execution (because it's too hard to do properly, quickly).

This works well, until:

- the remote user requests a critical file, such as `/etc/passwd`, or
- the remote user requests ``okfile ; rm -rf / &''

Security problems with CGI scripts, *continued*

A common technique is to run a site's web-server "as" a user account with very few privileges – LINUX systems often choose the *nobody* account.

If a web-server is attacked by exploiting a CGI script, the *nobody* account can still email a password file to a remote site, list internal host names and their addresses, etc.

A further problem with CGI scripts is that many standard scripts, distributed with free web-servers, have been known to introduce site vulnerabilities.

While these are usually corrected before the *next* distribution, attackers will frequently "scan" a web-server looking for a weak CGI script.

Examining a typical web-server's logfile, or error file will reveal hundreds of such scans per day:

```
[Mon Oct  4 14:41:34 2004] [error] [client 138.89.75.153]
    script not found or unable to stat:
        /home/www/cgi-bin/formmail.cgi
```

A well publicised case in late 1998 was a CGI script on *hotmail.com*. A flawed CGI script permitted unauthorized individuals to break into user's e-mail accounts and read their mail.

Web cookies

Cookies extend the Hyper-Text Transfer Protocol (HTTP) to allow a web-server to store information on the client for later retrieval. Using cookies allows *state information* to be maintained, making it easier to create services such as online shopping and site customisation for an individual.

Whereas a user would previously have to login on every visit, cookies provide session persistence, and allow this process to be transparent to the user.

Cookies were first introduced by Netscape in 1996 (with Navigator 1.1), and their specification has remained relatively unchanged. RFC 2109 attempts to formalise the cookie specification.

A cookie is simply a small piece of text (up to 4KB), stored as a *name=value* pair.

Cookie are "created" when a web-server sends a `Set-cookie` HTTP MIME header attached to a response for an HTML document or GIF image. For example:

```
Set-Cookie: ID=12345; expires=31-Oct-2004 14:12:40 GMT;
    domain=www.example.com; path=/; secure
```

Web cookies, *continued*

Once set, it is up to the user's browser to return it whenever a matching document is accessed.

For a cookie to be returned, the URL of the document must match the hostname (or domain) and path of the cookie, and the cookie must not have expired.

Only the *name=value* pair of the cookie is returned by the browser, in HTTP request header. From our example, the following HTTP header is added:

```
Cookie: ID=12345
```

This is added to any document requested from the machine `www.example.com`, and our example requires future requests to arrive via a secure HTTPS connection.

Keep in mind that a client has full access to the cookie, may read it, and possibly modify it before it is returned.

Are cookies a security concern?

In short no; cookies are only small pieces of *data* that are not executed by the browser, or the client's machine.

Despite popular media articles, cookies cannot read your disk files, scan your email, nor (by themselves) transmit your credit card details.

However, it's generally the *privacy* aspects of cookies that raise the most discussion.

The most familiar example of cookies eroding privacy is with banner advertisement companies such as `Doubleclick.com` Companies often attach cookies to their images (send them in the same response).

Whenever a user visits a page containing a banner from that company, the cookie is returned with the address of the page being viewed (in the `HTTP_REFERRER` field). By examining what pages a user accesses and when, the banner company is able to build up an online profile of the user.

This would typically include the pages the user visits, the advertisements that have been given to the user in the past, and which advertisements (if any) the user has clicked.

A special case of using cookies are *web-bugs*, simply cookies associated with 1x1 transparent GIF images, embedded on web-pages where advertising would seem inappropriate. As with advertisements, companies will pay websites small amounts of money to distribute their web-bugs.

Security aspects of cookies

While cookies themselves are harmless character strings, and may erode the privacy of web users, it is their actual *use* or contents that may *expose* a client's information to eavesdroppers, or permit an attacking client additional access rights to a web-server.

As a rule, a cookie should not actually *contain* state data between client and server, but should be used to *index* the data, typically in a database on the server machine.

Moreover, an untrusted system-administrator on the server could simply log all cookies returned to the web-server.

Extremely poor examples, often cited, include web-servers embedding names, passwords, and shopping-cart totals in cookies, with weak enshrouding of the contents.

Solution: a common requirement to keep a string on a client's machine in a form that can't be read. We (the server's owner) can keep a symmetric encryption key on our web-server to encrypt the cookie before sending it to the client; decrypting the cookie when it is returned.

A good practice is also to send and retrieve an additional cookie which is a digital signature of the data cookie. This helps the server detect tampering with the cookie.

Browser cookie implementation issues

Given the power that cookies provide, it was perhaps inevitable that browser bugs would appear. One such problem is deciding upon the domains for which a cookie can be set. This feature was intended to allow two sites within a domain (for example `www.bigbucks.com` and `images.bigbucks.com`) to access the same cookie. However, deciding what constitutes a domain is problematic.

A domain for a cookie is specified starting with a dot, to indicate that it needs only to match the end of the hostname rather than being an exact match to the entire name. Continuing the example above, specifying `domain=.bigbucks.com` would match both `www.bigbucks.com` and `images.bigbucks.com`.

Clearly this example is perfectly valid, but if unchecked it could be possible to specify cookies matching the entire `.com` top-level domain. We now have the specification:

Any domain that falls within one of the 7 top level domains (`"COM"`, `"EDU"`, `"NET"`, `"ORG"`, `"GOV"`, `"MIL"`, and `"INT"`) only require 2 dots. Other domains require at least 3.

An exploit became known mid-2000 which showed how it was possible to access any cookie stored on a user's machine running any version of Internet Explorer up to and including version 5.0.

Explorer treated the URL `www.test.com%2F.amazon.com` as if it comes from `amazon.com`, even though `%2F` is equivalent to the `/` character. By embedding a URL like this in a page, a malicious webmaster could access personal information either from the cookie itself or by impersonating the user at a site like `amazon.com` that stores personal information.

Automatic Directory Listings

Some "out-of-the-box" web-server configurations permit full directory browsing in the absence of files such as *index.html*, or always prepend files such as *README* to the beginning of any directory listings.

The automatic directory listings provided by the Apache, MS-IIS, and Netscape servers, for example, are convenient, but have the potential to provide an attacker with access to sensitive information.

For example, automatic directory listings *do* try to hide certain files, perhaps ones with critical extensions such as *.cgi* or those with certain file-permissions.

However, consider

- text-editors' automatic backup files containing the source code to CGI scripts,
- source-code control logs,
- symbolic links, once created for your convenience and not removed,
- directories containing temporary files, etc.

Of course, even turning off automatic directory listings doesn't prevent attackers fetching files whose URLs and names they can guess.

Server-Side Includes

SSIs (Server-Side Includes) are directives that may be placed in HTML pages, to be evaluated on the web-server while the pages are being served.

SSIs add dynamically generated content to an existing HTML page, without having to serve the entire page via a CGI program, or module (such as PHP).

Typically, files with the extension of *.shtml* are considered (configured) to contain SSI requests.

Examples include:

- reporting today's date:

```
Today is <!--#echo var="DATE_LOCAL" -->
```

- reporting a file's modification time:

```
This page modified <!--#flastmod file="index.html" -->
```

- or running general programs:

```
<!--#include virtual="/cgi-bin/counter.sh" -->
<!--#exec cmd="ls" -->
```

Security problems introduced by Server Side Includes include:

- denial-of-service attacks, where an incorrectly (lazily) configured server parses *all* pages looking for SSIs, and
- modified access privileges, for users local to the web server itself. SSI-enabled files can execute any CGI script or program under the permissions of the user and group of the web-server itself. Thus *any* user-written script can run with these privileges.

Protecting Local Files from Remote Clients

An important security aspect of common web-servers is their definition of *default file access*.

Most servers take a permissive attitude to file distribution, allowing any document in the document root to be transferred unless it is specifically forbidden.

Unless steps are made to change it, if the server can find its way to a file through normal URL mapping rules, it serves them to clients.

Consider a poorly configured web-server, or one with which an intruder has modified the configuration. An intruder (as root) simply running the command:

```
root> cd / ; ln -s / WWW
```

Allows them to later browse the readable parts of the whole file system with the basic URL `http://machine.com/` In general, such casual web-browsing would not be considered suspect, or quickly identified.

The configuration of modern web-servers permits browsing to be permitting or denied on a per-directory basis:

```
<Directory />
     Order Deny,Allow
     Deny from all
</Directory>

<Directory /home/*/*/WWW>
     Order Deny,Allow
     Allow from all
</Directory>
```

Constrained Access with *.htaccess*

Confi guration controls and commands *must* permit fi ne granularity of access control to pages and data that reside on the server.

This can generally be performed from either system-wide or per-directory confi guration fi les, often named *.htaccess*

Examples include:

- Restriction of the HTTP operations that may be performed:

```
<Limit GET POST>
    require valid-user
</Limit>
```

- Restriction by IP address, subnet, or domain .

```
order deny,allow
deny from all
allow from .uwa.edu.au
allow from 138.65.2
```

Subject to the traditional problems with IP-address or domain-name spoofi ng, unless a web-server is run behind a fi rewall (perhaps in a DMZ) that makes some effort to detect and reject spoofi ng.

Restriction by IP address can be "confused" if clients frequently visit via a web-proxy machine, in which case access must be permitted from the proxy (the client's IP address cannot be determined). In this case, trust is shift to the proxy's settings, to ensure that it will only proxy for trusted clients.

User-Authentication with *.htaccess*

Access to directories (and their subdirectories) may also be constrained to certain user-names and group members.

```
<Directory /fullpathname/directory>
    AuthUserFile  /WWW/secure/.htpasswd
    AuthGroupFile /dev/null
    AuthName      ByPassword
    AuthType      Basic
```

A server can further strengthen the limitations of a shared password fi le by using the require directive, as with:

```
    require user chris
    require group staff
</Directory>
```

However, because modern web-servers try to disassociate themselves from their host operating system, the lists of users and groups are not "real" ones, but ones that must be maintained by additional programs:

```
htpasswd -c .htpasswd username
Adding password for username.
New password:
(not echoed) password
Re-type new password:
(not echoed) password
```

Furthermore, unlike an operating system's login session, in which the password is passed over the Internet just once, a browser sends these passwords every time (unencrypted) it fetches a protected document.

Web-server Logging and Privacy

All requests for web-served documents may be logged by the Web server. Typical details include the client's hostname (or IP on a harried server), date and time, HTTP request, URL, size and response code:

```
  csse2722 - - [01/Oct/2004:11:33:19] "GET /run/help317?a=162...
```

Clients' names are also logged if *.htaccess* authentication was used to access a URL.

As most web-client requests arrive from individual machines, if not through a proxy, web logfi les invade users' privacy. Some servers may log the URL being viewed (your *referer* page) at the time you requested the new URL. Of course, all of these access details are only kept for statistics generation and debugging.

However, there exist several vulnerabilities:

- some sites may leave the logs open for casual viewing by local users,
- the logfi les are created and owned by the same userid as the web-server itself. This permits a successful attacker to view or truncate the logs.
- the contents of queries using the HTTP GET request appear in the server log fi les because the query is submitted as part of the URL (queries using POST are not so logged)

Although web logfi les can only log requests made *to* a web-server web proxies also log requests made of them, with very similar details. The web-browsing habits of individuals from *within* a company, including the popular pastimes of job-hunting and bird-watching, may be tracked.